On The [Ir]relevance of Network Performance for Data Processing

Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle,
Radu Stoica, Bernard Metzler, Ioannis Koltsidas, Nikolas Ioannou
IBM Research, Zurich

1 Introduction

Modern data processing frameworks are used in a variety of settings for a diverse set of workloads such as sorting, indexing, iterative computations, structured query processing, etc. As these frameworks run in a distributed environment, a natural question to ask is – *how important is the network to the performance of these frameworks?* Recent research in this field has led to contradictory results. One camp advocates the limited impact of networking performance on the overall performance of the framework [16]. On the other hand, there is a large body of work on networking optimizations for data processing frameworks [9, 10, 18, 19, 21].

In this paper, we search for a better understanding of the matter. While answering the basic question concerning the importance of the network performance, our analysis raises new questions and points to previously unexplored or unnoticed avenues for performance optimizations. We take Apache Spark [2] as a representative of a modern data-processing framework. However, to broaden the scope of our investigation, we also experiment with other frameworks such as Flink, PowerGraph or Timely. In our study – rather than analysing Spark-specific peculiarities – we look into procedures and subsystems that are common in any of these frameworks such as networking IO, shuffle data management, object (de)serialization, copies, job scheduling and coordination, etc. Nonetheless, we are aware that the roles of those individual components are different for the various systems, and we exercise caution when making generalized statements about the performance.

Our study reveals three main findings: (a) up to a certain level, the performance of the network has significant effects on the overall performance of the data processing framework. Specifically, for all the workloads and frameworks we analyzed, moving from a 1 to a 10 Gbps network reduced the query response time by a factor of two and more (see Figure 1). This result directly refutes the statement about the importance of network performance made in a previous study [16]; (b) moving from a 10 to a 40 Gbps network results in almost no performance gains; (c) the inability of the data processing frameworks to leverage network speeds higher than 10 Gbps is caused by a high CPU footprint that prevents the system from balancing compute and I/O. For instance, in the case of Spark, the high amount of CPU work done per byte moved eclipses any potential gains coming from the network. We investigate a number of ways to balance CPU and network time and discuss possible avenues for data processing frameworks to perform better on upcoming fast networks.

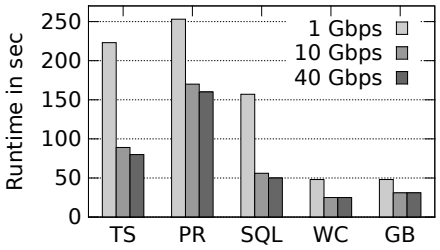


Figure 1: Runtimes for various workloads (see Table 1) on a 1, 10, and 40 Gbps network.

2 Does Network Performance Matter?

A recent study from UC Berkeley concluded that network optimizations can only reduce a job completion time by 2% [16]. This polarizing statement remains the last known ground truth about the Spark networking behavior and is being picked up by others as well [13]. We start our investigation by verifying this statement. Instead of speculating about the effect of networking performance on job completion time using system instrumentation, we actually run identical workloads on the three generations of networks – 1, 10, and 40 Gbps – with an otherwise identical system configuration. We use a

Symbol	Workload	Dataset
TS	The TeraSort bencmark [6]	240 GB Key-Value pairs (2.4B tuples of 100 bytes)
PR	The Page Rank algorithm (16 iterations)	The Twitter graph (41.7M nodes, 1.47B edges) [12]
SQL	20 Spark SQL TCP-DS queries	TPCDS dataset with Scale Factor =1000 (1 TB dataset, 300 GB compressed)
WC	Word Count	The Google Ngram Corpus 28 GB [7]
GB	Group By	80 MB dataset (10M keys of 8 bytes)

Table 1: Workload description.

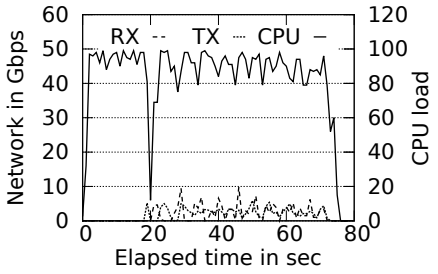


Figure 2: CPU and network profile (40 Gbps) of an executor for a TeraSort run.

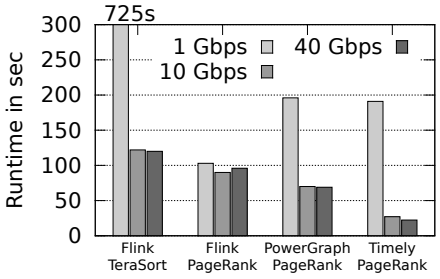


Figure 3: Effect of 1, 10, and 40 Gbps networks on Flink [1], PowerGraph [11, 5], and Timely [8].

13-server cluster with dual socket Xeon E5-2690 CPUs, 128 GB of DDR3 DRAM, Intel DC S3700 SSDs for storage, and Intel I350 (1 Gbps), Chelsio T4 (10 Gbps), and T5 NICs (40 Gbps). One of the 13 machines is used for running the HDFS namenode, YARN resource manager, and the Spark driver. We run multiple workloads which are described in Table 1. For all the workloads, Spark (2.0 from GitHub) uses 1 executor per machine (12 in total) and 64 GB of memory. To eliminate disk related overheads, the remaining 64 GB of RAM is mounted as the `temp` directory (`hadoop.temp.dir`) for YARN and Spark. In all experiments, HDFS always runs on the 40 Gbps network and the Spark numbers reported are from a warm HDFS cache run which involves no disk IO.

Figure 1 summarizes our results. The y-axis shows the job run time with three different networks. As shown by our experiments, without exception, all workloads see shorter job runtimes when moving from 1 to 10 Gbps. However, the gains are marginal when moving from a 10 to a 40 Gbps network. Figure 2 shows the CPU and network profile of one executor during an execution of TeraSort. As illustrated, during the shuffle phase (between the 20-78 second marks), the receive network bandwidth *peaks* around the 10 Gbps mark while the transmit bandwidth does not go beyond 6-7 Gbps. The CPU utilization remains high between 80-100%. The execution profiles of other workloads show a similar pattern where the peak utilized network bandwidth is always above 1 Gbps but below the 10 Gbps mark.

At this point we became curious if other data processing systems exhibit the same behavior. To answer this question, we ran TeraSort and PageRank experiments on a variety of systems such as Flink, PowerGraph and Timely. We chose TeraSort and PageRank as those were the common workloads we could run on any of the systems. The aim of these experiments is not to compare the relative performance (as they can be optimized by system-specific settings) but to see the relative gains that they get from using 1, 10, and 40 Gbps networks. Figure 3 shows our results, which are very much in line with the previous Spark experiments.

We conclude therefore that, in contradiction to the previously reported assumption that "network I/O is mostly irrelevant to the overall performance, even on 1Gbps networks" [16], networking performance does matter. In our setup, moving from a 1 Gbps to a 10 Gbps network resulted in significant performance gains and a reduction of the response time by a factor of $1.5-2.5\times$. The performance gain is also visible for workloads which are not necessarily thought to be network bound such as PageRank (where the shuffle data is significantly smaller than the input data size) and Spark SQL queries (which operate on compressed data).

3 Why 40 Gbps Networks Do Not Help?

A natural question to ask following the last analysis is *what stops these frameworks from leveraging a 40 Gbps network?*. The answer lies in the CPU usage of these frameworks. For our analysis, we take Spark as a representative data processing framework due to both its popularity and general applicability. We start by first breaking down the CPU usage for the TeraSort workload. We chose TeraSort as it is one of the most well known workloads and because of the straightforward nature of the network I/O involved. A TeraSort run consists

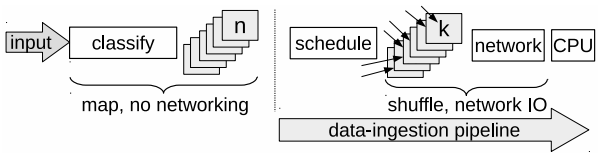


Figure 4: A view of the Spark data ingestion pipeline.

of two phases. The first phase is a *mapping* or classification phase - where individual workers read their part of the key-value (KV) input data and classify the KV pairs based on their keys. This phase involves only very little networking as all workers run locally on the nodes that host the input HDFS data blocks. This assumption can be verified by looking at the networking traffic in Figure 2 between the 0-20 second marks. The classified output data (or shuffle data in Spark terminology) is stored in the memory-resident `temp` directory. During the second so called *reduce* phase, each worker collects all KV pairs from all workers for a particular key range, and then sorts the data.

Two features of TeraSort are relevant for our analysis: (a) in the reduce phase all nodes communicate to all other nodes; (b) the amount of shuffle data is the same as the input data. It has been pointed out by Ousterhout et al. that networking researchers often inflate the importance or the size of the shuffle data in comparison to the overall input size (in [16], Section 4.4). TeraSort workload does not suffer from such a bias and, hence, should be an ideal candidate to be accelerated by a fast networking. However, as we have shown in the previous section, the TeraSort response time reduces by half when moving from a 1 to a 10 Gbps network (which in itself leaves a great margin untapped on the table), but stays almost unaffected when moving from a 10 to a 40 Gbps network.

3.1 CPU Time vs Network Time

To understand Spark's processing cost we profile its CPU usage. Figure 4 shows a simplistic view of TeraSort which helps us understand Spark's CPU requirements, especially for the reduce phase of TeraSort. The data ingestion pipeline of the reduce phase involves scheduling, fetching data over the network, and finally sorting. This pipeline runs on all cores in multiple waves of tasks on all the compute nodes. In the follow up discussions we calculate the *data-ingestion* bandwidth of this pipeline for various settings. Naturally, the performance of such a pipeline depends upon both the network as well as the CPU performance, which together *should* dictate the overall job run time.

The CPU breakdown (see Table 2, discussed in more detail in Section 3.3) reveals that Spark is CPU bound and that the majority of the CPU cycles are spent on sorting. In absolute terms it takes 1.2 seconds to sort approximately 128 MB of KV data. In comparison, it takes 1110 ms, 110 ms, and 27 ms (theoretical calculations) to fetch 128 MB of data over a 1, 10, and 40 Gbps network respectively. Therefore, the relative importance of the network performance in comparison to the CPU time (which stays constant) decreases from almost 93% to 9.3%, and then to just 2% when moving from a 1 to a 10 and 40 Gbps network. Consequently, the CPU performance lags behind the raw networking performance (ignoring other implementation related overheads which we discuss in Section 3.3) by 1-2 orders of magnitude. Hence, between 10 and 40 Gbps, due to the heavy CPU usage, the relative network performance improvements of 4× will result in a modest 5% improvement in the end-to-end job performance. With such performance figures, the Spark single-core data ingestion pipeline for TeraSort calculates to 834.5 Mbps/core and would require 48 cores to fetch and process data at a rate of 40 Gbps.

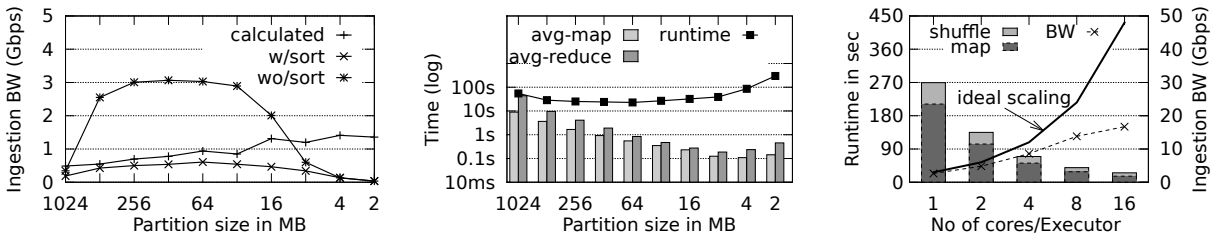
To conclude, the current Spark implementation runs in a heavily unbalanced configuration where the CPU time overshadows any networking performance gains. It is therefore expected that an uneven system cannot deliver the best performance available when utilizing all its resources [20, 14].

The next question is if the CPU time and networking time can be brought into balance. In the rest of this section we explore three orthogonal ways of increasing network utilization: a) we reduce the complexity of the shuffle operations by reducing the partition size; b) we explore whether the CPU overhead can be significantly reduced by optimizing Spark; and c) we investigate whether increasing the number of cores available per network link can lead to an increase in performance.

3.2 Effect of Partition Size

One obvious solution to improve the ratio of CPU to network time is to reduce the complexity of the shuffle operations in Spark. The high-level idea is that by reducing the partition size, the sorting time – which follows a $O(n \log n)$ time complexity – becomes comparable to the network fetch time, which decreases linearly with the partition size. Therefore, we expect the ratio of CPU to network time to improve by using smaller partitions.

To confirm our hypothesis, we vary the TeraSort partition size and show the execution time in Figure 5a. In this setup we chose to run a smaller 12 GB data set with 12 executor with 1 core each. This was necessary as a 240 GB dataset using 1 MB partitions generated 240k tasks which turned the scheduler into a major performance bottleneck. The three lines plotted in Figure 5a are: (a) the theoretical ingestion bandwidth predicted from the partition size; (b) the measured ingestion bandwidth when sorting the dataset(*w/sort*); (c) the measured ingestion bandwidth with sorting disabled (*wo/sort*). As



(a) Partition size vs data ingestion bw.

(b) Partition size vs runtime.

(c) Effect of core scaling.

Figure 5: Profile of TeraSort.

expected, a smaller partition size does help to improve the theoretical bandwidth, however, the system is marred by implementation related overheads, and does not go above 606 Mbps/core. It is worth noting that the *wo/sort* bandwidth peaks at only 3 Gbps/core, which represents the maximum per-core data ingestion rate for the system.

In Figure 5b we plot the effect of the partition size on the total runtime of the sorting job and also include the average map and reduce times. As observed earlier, smaller partition sizes help reduce the job runtime from 54.7 seconds for 1 GB, to 22.9 seconds for 64 MB. However, further decrements in the partition size do not result in runtime improvements despite reducing the runtime of the individual map/reduce tasks. This is because a higher number of partitions generates a higher scheduling overhead and increases the number of output segments each shuffle task needs to assemble. The inefficiency of handling small partitions indicates a practical limit to the per-task partition size which inhibits the framework to scale to smaller data set sizes. This behavior was also pointed out by McSherry et al. who showed that modern data processing frameworks use CPU/IO resources very inefficiently [14]. Tiny-Tasks [15] made a case for small tasks operating at granularities of milliseconds. However, (a) their focus has been more on the scheduler design [17] than on the whole system optimization; (b) their assumed networking latencies are still too high ($\sim$100 ms) for a modern rack-scale deployment.

To conclude, smaller partition sizes do not necessarily result in shorter runtime due to excessive amounts of scheduling, shuffling, and implementation-related overheads found in Spark.

3.3 CPU Profile

We next investigate where CPU time is consumed during a Spark TeraSort run to establish if improving a specific Spark component can lead to a significant improvement in performance.

The execution profile of TeraSort, shown in Table 2, assigns CPU cycles to the various Spark components involved in the execution. For the map phase, 79% of the execution time is spent in the Spark framework, 9% in the JVM, and 12% in the kernel (i.e., memory management, network, copying). If we zoom in further, 26% of the execution time is spent on (de)serialization and data copy within the JVM, 24% on reading/writing from/to HDFS and calculating checksums, 16% is spent on sorting the data (by the Spark Shuffler), and 13% in other Spark routines (e.g., iterator operations). Looking at the reduce phase, 45% of the time is actually spent on sorting the data, the rest of the time is consumed by 14% iterator related routines, 7% (de)serialization and copy operations. The JVM and kernel execution times are similar to the map phase.

To further identify bottlenecks in the data ingestion pipeline we remove sorting and only gather the unsorted data for counting (we call this benchmark TeraCount). The theoretical bandwidth of such a data ingestion pipeline should be limited by the performance of the network. However, we found out that the current Spark execution is still bound by the CPU. The last column of Table 2 shows the CPU profile breakdown for the reduce phase of TeraCount (note that the map phase is similar to TeraSort's). As can be seen, up to 52% of the CPU cycles are spent in iterator related routines. Spark makes extensive use of iterators as a construct to implement a lazy execution model, which is a key property of the Spark runtime. For example, TeraCount has at least 7 iterators stacked on top of each other (fetching, compression, deserialization, aggregation, etc.). Every tuple access moving through the data pipeline requires multiple `next/hasNext()` calls. Besides iteration, 17% of the time is spent on (de)serializing and data copying. A further 11% is spent on other unclassified Spark related routines. The remaining 20% of the time is spent in the kernel and the JVM on various memory, networking, and process management routines.

(De)serialization related overheads have been identified by Spark developers [3] and others as well [4]. Data copies (and memory cleaning) happen frequently for pointers when templated datasets are re-sized in the JVM. Consequently, the long code path executed by the Spark core engine for every block of data contributes significantly towards an unbalanced system with a dis-

		TeraSort		TeraCount
		map	reduce	reduce
Spark	(de)ser. + copy	26%	7%	17%
	HDFS rw + csum	24%	-	-
	Sorting	16%	45%	-
	Iterator	8%	14%	52%
	Misc	5%	12%	11%
	JVM	9%	6%	7%
	Kernel	12%	16%	13%

Table 2: The CPU profile of a Spark executor.

proportionate amount of CPU cycles requirements. For example, under one setting in the Spark network receive pipeline, processing a block of 80MB shuffle data takes over 2 seconds, where a significant amount of time is spent on reassembling network data into Spark frames and messages (no deserialization or data copying is happening here). In contrast, receiving the actual data on a 40 Gbps network only takes 16 ms.

Our analysis shows that the Spark's inefficiencies are not located in a single component or layer that can be targeted by local optimizations. Rather, the high CPU inefficiency is a consequence of the architecture of the framework and requires deeper cross-layer optimizations. Improving a single component will simply move the bottleneck to the next stage in the data pipeline.

3.4 Core Scaling

Another tempting way of improving the CPU to network ratio is to simply add more cores per network link. The increase in CPU power can be obtained either by investing in more expensive multi-socket servers or by further increasing the number of cores per CPU.

To validate the feasibility of this approach, we run an experiment where we vary the number of cores while processing the same amount of data per-machine. For example, while running TeraCount on 240 GB using 12 machines, each machine processes 20 GB of data with a variable number of cores (between 1 to 16). In Figure 5c we plot the execution times and calculate the data ingestion bandwidth from the execution time of the reduce phase. Spark scales well between 1-2 cores and closely follows the *ideal scaling* line. The ideal scaling is computed by taking the 1 core performance (3 Gbps) and multiplying it with the number of cores. Beyond 2 cores, the actual measurements start to divert from the ideal curve. With ideal scaling, the Spark executor should have been close to saturating the 40 Gbps link, instead we measure a peak bandwidth for Spark that is below the 20 Gbps mark.

The results suggest that a part of the TeraCount computation cannot be efficiently parallelized. As a back-of-the-envelope calculation, we apply Amdahl's law to identify the serial and parallel computation components of TeraCount. We find that counting 240 GB of data involves a serial execution component of 9 seconds and a parallel component of 260 secs, irrespective of the number of cores used. Although the serial execution component seems relatively small compared to the total runtime (only 3.75%), it fully explains Spark's sub-optimal scaling with the number of cores and limits how much the data ingestion bandwidth can be accelerated given a fixed problem size. The implications are that, in order for Spark to fully benefit from additional cores, all serial execution components need to be identified and removed, even if they represent a relatively small part of the overall execution time.

4 Conclusions and Outlook

In summary, our investigation so far has shown three things. Firstly, faster networks can improve the overall data processing performance as long as there are enough CPU cycles available. Secondly, Spark as of now cannot take full advantage of 40Gbps networks as the data ingestion pipeline becomes CPU bottlenecked. Thirdly, the CPU to network imbalance cannot be changed through optimizations such as reducing the partition size, optimizing a single Spark component, or just by adding more cores per network link.

We argue that deeper architectural changes are required in order for Spark to leverage faster networks. We are in a micro-second era where both networking and storage devices have latencies in 10-100s of microseconds. The raw device performance will continue to improve, making the thick software layers (and the associated CPU overhead) a performance bottleneck. Spark's inability to leverage high-performance networks is an example of this problem. A millisecond here and there (RPCs, scheduling, serialization, etc.) quickly adds up to several seconds in the total execution time. While those inefficiencies have very little impact on the overall system performance in the case of a 1 Gbps network, they are now being exposed on fast 40 Gbps networks.

The CPU heavy operations in Spark are a reminder of the struggles Linux had to keep up with the emergence of non-volatile storage. The block and file system layers which were designed and optimized for the milliseconds-to-seconds timescales suddenly found themselves running on a microsecond timescale. Consequently, some of the inefficiencies in these software stacks got exposed and performance became CPU bound. However, over the years multiple optimizations have been put in place to make the stack thinner, slicker, and more CPU efficient. The same needs to be done with Spark, and in-general with any distributed computing framework, in order to make further inroads toward μ*second-scale data processing*.

References

[1] Apache Flink 0.10.2, Scalable Batch and Stream Data Processing, https://flink.apache.org/.

[2] Apache Spark home page, http://spark.apache.org/.

[3] Introducing Spark Datasets, https://databricks.com/blog/2016/01/04/introducing-spark-datasets.html.

[4] Juggling with Bits and Bytes, https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html.

[5] PowerGraph: A framework for large-scale machine learning and graph computation, https://github.com/dato-code/PowerGraph.

[6] Sort Benchmark Home Page, http://sortbenchmark.org/.

[7] The Google Books Ngram Viewer Dataset, http://storage.googleapis.com/books/ngrams/books/datasetsv2.html.

[8] Timely Dataflow, https://github.com/frankmcsherry/timely-dataflow/.

[9] CHOWDHURY, M., ZAHARIA, M., MA, J., JORDAN, M. I., AND STOICA, I. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada, 2011), SIGCOMM '11, pp. 98–109.

[10] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA, 2014), SIGCOMM '14, pp. 443–454.

[11] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA, 2012), OSDI'12, pp. 17–30.

[12] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 591–600.

[13] MASCHHOFF, K. J., AND RINGENBURG, M. F. Experiences running and optimizing the berkeley data analytics stack on cray platforms, https://cug.org/proceedings/cug2015_proceedings/includes/files/pap142.pdf. In *Cray Usergroup Meeting* (2015).

[14] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland, 2015), HOTOS'15, pp. 14–14.

[15] OUSTERHOUT, K., PANDA, A., ROSEN, J., VENKATARAMAN, S., XIN, R., RATNASAMY, S., SHENKER, S., AND STOICA, I. The case for tiny tasks in compute clusters. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems* (Santa Ana Pueblo, New Mexcio, 2013), HotOS'13, pp. 14–14.

[16] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA, USA, 2015), NSDI'15, pp. 293–307.

[17] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania, 2013), SOSP '13, pp. 69–84.

[18] PRAKASH, P., DIXIT, A., HU, Y. C., AND KOMPELLA, R. The tcp outcast problem: Exposing unfairness in data center networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA, 2012), NSDI'12, pp. 30–30.

[19] RASMUSSEN, A., LAM, V. T., CONLEY, M., PORTER, G., KAPOOR, R., AND VAHDAT, A. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California, 2012), SoCC '12, pp. 13:1–13:14.

[20] RASMUSSEN, A., PORTER, G., CONLEY, M., MADHYASTHA, H. V., MYSORE, R. N., PUCHER, A., AND VAHDAT, A. Tritonsort: A balanced large-scale sorting system. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, 2011), NSDI'11, pp. 29–42.

[21] ZHANG, J., ZHOU, H., CHEN, R., FAN, X., GUO, Z., LIN, H., LI, J. Y., LIN, W., ZHOU, J., AND ZHOU, L. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA, 2012), NSDI'12, pp. 22–22.

Notes: IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Other products and service names might be trademarks of IBM or other companies.